# SGEMM Test in Tesla P4

WENQINGQIAN

*Compiled April 23, 2024*

**This article records my implementation and experimental results of the matrix multiplication algorithm. Includes comparison with high-performance implementation provided by cuBLAS library**
**By optimizing the organization of thread blocks, the use of shared memory, and data access patterns, the computational efficiency of matrix multiplication has been successfully improved.**
**Code see Github**
**Profile see here**

## 1. ENVIRONMENT

- CUDA: 12.2
- compiler option: -O3 –use_fast_math

**Table 1.** Tesla P4

| GPU Config | Value |
|---|---|
| FP32 Core Num | 128 |
| SM Num | 20 |
| ShareMem Size | 49152 |
| FP32 Peak Performance | 5701.12 GFLOPS |

## 2. EXPERIMENT

The algorithm used is based on openmlsys

### SetUp

$$GEMM = \alpha A \times B + \beta C \tag{1}$$

$$A \in \mathbb{R}^{M\times K}, B \in \mathbb{R}^{K\times N}, C \in \mathbb{R}^{M\times N}$$

$$M = 2560, N = 2560, K = 1024, Iter : 50$$

### A. GEMM tile

As shown in the figure 1, a thread is responsible for a row in A and a column in B, and calculates the value of the red area. Each row and column here is of float4 type.

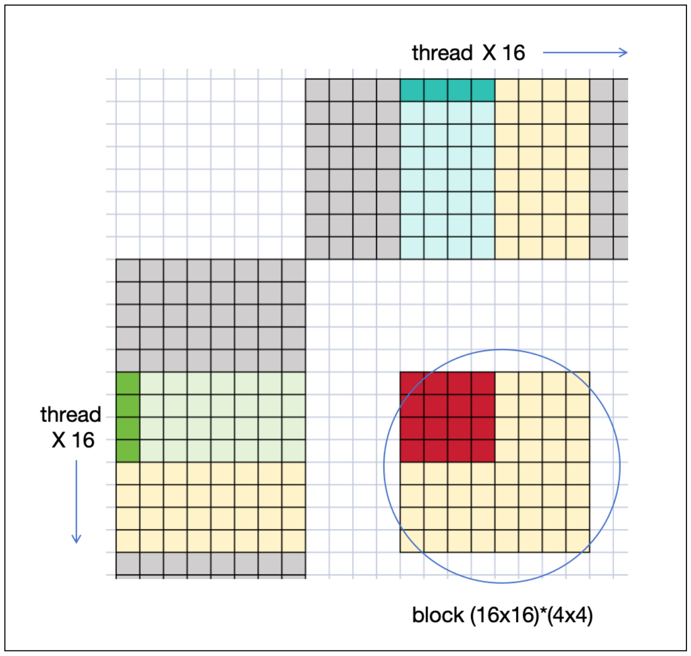


**Fig. 1.** tile

### B. GEMM shared memory

Reading Thread layout

- A : RowMajor
- B : ColMajor

Shared Memory layout

- A : RowMajor
- B : RowMajor

**Algorithm 1.** gemm tile

```
1: Input: block<16,16>
2: A rowOffset to each thread in each block
3: B colOffset to each thread in each block
4: for k in range(K) do
5:     for i in range(4) do
6:         result[i] += A'[k][i] * B'[k]          ▷ float4 type
7: result *= $\alpha$
8: C = C * $\beta$ + result
```

As shown in figure 3 ,when reading the data from global mem to shared mem, one thread processes 4 (A:2,B:2) float4 at the same time

For reading B into shared memory in openmlsys, B is divided into two parts along the row direction, and threads are arranged row-first. In this method, cutting B into two parts along the column direction will greatly improve performance, from 45% peak to 69% peak.

**Algorithm 2.** gemm shared memory

```
1: Input: block<16,16>, Tile: 16
2: A rowOffset to each thread in each block
3: B colOffset to each thread in each block
4: for range(K/16) do
5:     load A′[x][y], A′[x+64][y] to Shared Mem [128,4]
6:     load B′[x][y], B′[X][y+16] to Shared Mem [16,32]
7:     for range(16) do
8:         for range(4) do
9:             result += A′ * B′
10: result *= $\alpha$
11: C = C * $\beta$ + result
```

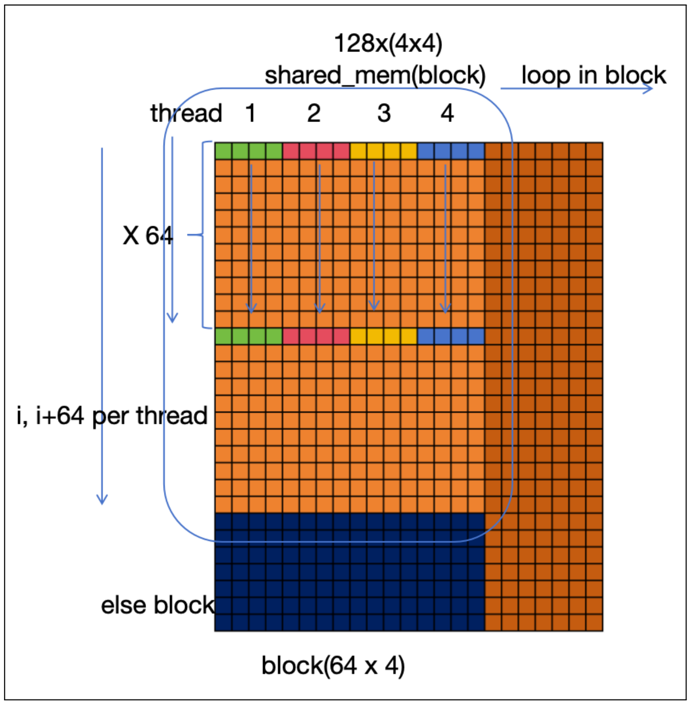


**Fig. 2.** shared mem reading A

The calculation is consistent with GEMM tile. The difference is that each thread will process the data of the current position and the data of the position after a fixed value interval as shown in figure 4

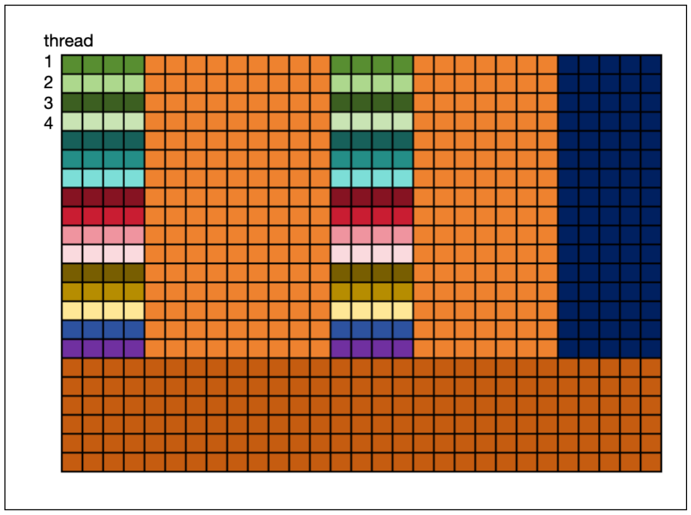


**Fig. 3.** shared mem reading B

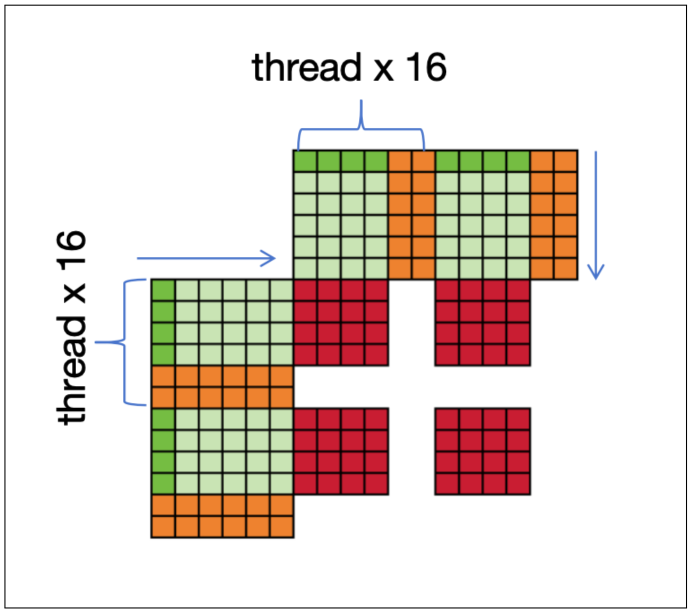


**Fig. 4.** shared mem computation

### C. GEMM shared memory with ColMajor Smem A

Reading Thread layout

- A : RowMajor
- B : RowMajor

Shared Memory layout

- A : ColMajor
- B : RowMajor

In this method 3, cutting B into two parts along the row direction as shown in figure 5, which can make the reading thread layout of B the same as the computation layout

For the data in A, when calculating, the thread reads column-first, and when storing, it accesses by row due to float4. This process requires constructing each float4 separately.

In the previous algorithm, the data is directly **stored** in Shared Memory in row-first order, and is constructed into column-form

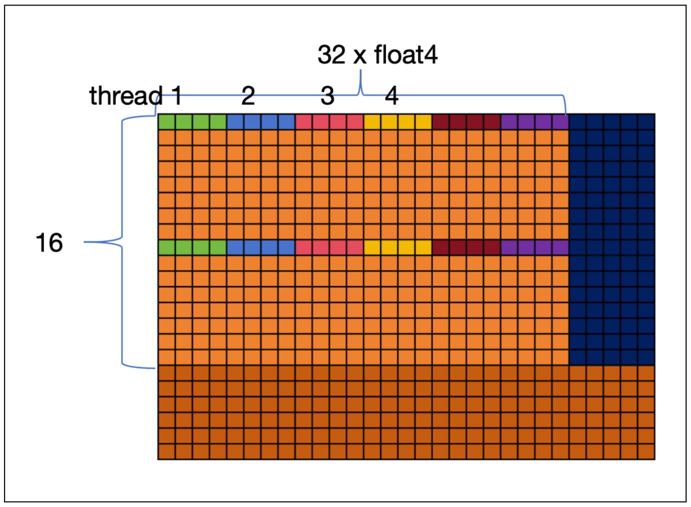


**Fig. 5.** shared mem reading B Cut CowMajor

fragments when **reading**. This method transpose the matrix A when storing into Shared Memory, that is, the column-form fragments are performed during the **storage** process, and is obtained directly when **reading** as shown in figure 6

**Algorithm 3.** gemm shared memory with ColMajor Smem A

```
1:  Input: block<16,16>, Tile: 16
2:  A rowOffset to each thread in each block
3:  B rowOffset to each thread in each block
4:  for range(K/16) do
5:      load A′[x][y], A′[x+64][y] to Shared Mem [16,32]
6:      load B′[x][y], B′[X+8][y] to Shared Mem [16,32]
7:      for range(16) do
8:          load Shared Mem A′B′ to fragmentA′B′
9:          for range(4) do
10:             result += fragmentA′ * fragmentB′
11: result *= $\alpha$
12: C = C * $\beta$ + result
```

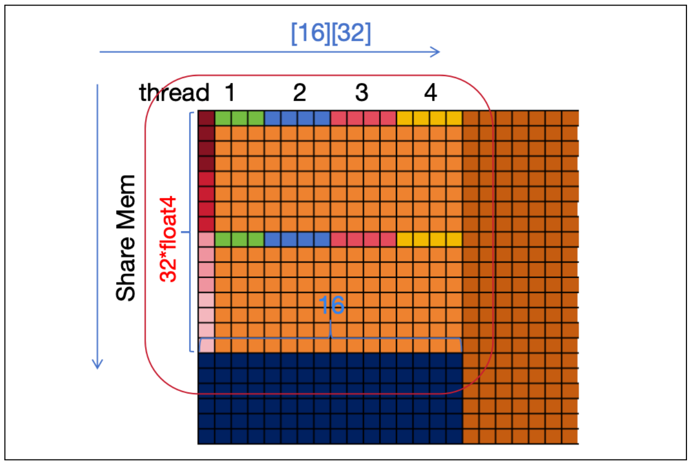


**Fig. 6.** transpose A

### D. GEMM hide shared memory IO latency

The instruction *LDS* is used in the GPU to read data from the shared memory. After this instruction is launched, it will not wait for the data to be read into the register before executing the next instruction. Only instructions that rely on the data read by the *LDS* instruction will wait for the completion of the read.

For the *for* loop on Line 7 in algo 6, we load shared mem to fragment and then execute the computation which will lead to the latency.

In this method 4, We are transforming the access to shared memory into a pipelined form by double buffer.

In this experiment this had only a minimal impact.

**Algorithm 4.** gemm hide shared memory IO latency

```
1:  Input: block<16,16>, Tile: 16
2:  A Offset to each thread in each block
3:  B Offset to each thread in each block
4:  for range(K/16) do
5:      load A′[x][y], A′[x+64][y] to Shared Mem [16,32]
6:      load B′[x][y], B′[X+8][y] to Shared Mem [16,32]
7:      load Shared Mem A′B′[0] to fragmentA′B′[0]
8:      for i in range(16) do
9:          load Shared Mem A′B′[i+1] to fragmentA′B′[(i+1)%2]
10:         for range(4) do
11:             result += fragmentA′[i%2] * fragmentB′[i%2]
12: result *= $\alpha$
13: C = C * $\beta$ + result
```

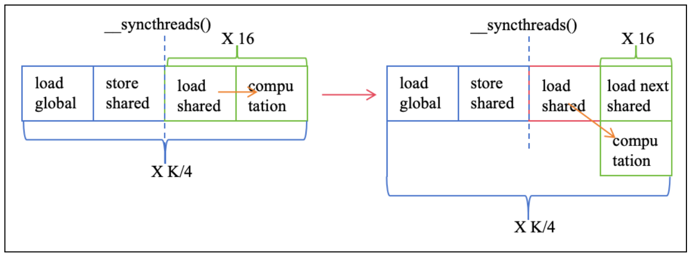


**Fig. 7.** hide shared memory IO latency

**Algorithm 5.** hide global memory IO latency

```
1: Input: block<16,16>, Tile: 16
2: A Offset to each thread in each block
3: B Offset to each thread in each block
4: load A′[x][y], A′[x+64][y] to Shared Mem [0][16,32]
5: load B′[x][y], B′[X+8][y] to Shared Mem [0][16,32]
6: load Shared Mem[0] A′B′[0] to fragmentA′B′[0]
7: stage = 1
8: for range($\mathbf{K}/16$) do
9:     A′ colOffset to next Tile
10:    B′ rowOffset to next Tile
11:    load A′B′ to buffer
12:    for i in range(15) do
13:        load Shared Mem[!stage] A′B′[i+1] to fragmentA′B′[(i+1)%2]
14:        for range(4) do
15:            result += fragmentA′[i%2] * fragmentB′[i%2]
16:    if i < $\mathbf{K}/16$ - 1 then
17:        load buffer to Shared Mem [stage][16,32]
18:        stage = !stage
19:    load Shared Mem[!stage] A′B′[0] to fragmentA′B′[0]
20:    result += fragmentA′[1] * fragmentB′[1]
21: result *= $\alpha$
22: C = C * $\beta$ + result
```

### E. GEMM hide global memory IO latency

In this method 5, We are transforming the access to global memory into a pipelined form by double shared memory.

As pointed by the purple arrow in the figure 9, we use computation or other operations to hide these latency.

## 3. DEBUG LOG

**threadIdx.** For two dimensional block, the unified thread ID = threadIdx.y * blockDim.x + threadIdx.x

**Register.** The __device_builtin__ data structure tensor2d have four **size_t** data member. When I optimized the perfromance by hiding global memory IO latency, the costs of register improved, which lead to the register spill and make the performance of **GEMM hide global memory IO latency** very bad which happend in Tesla P4, RTX 3080, RTX 4090, etc.

Change all **size_t** to **unsigned int**.

**CMake file.** Listing 2 will cause a delay of more than 100 seconds when the program starts. (unknown)

**Listing 1.**

```
project(MyProject CXX CUDA)
add_executable(gemm ${cudafile})
```

**Listing 2.**

```
project(MyProject)
cuda_add_executable(gemm ${cudafile})
```

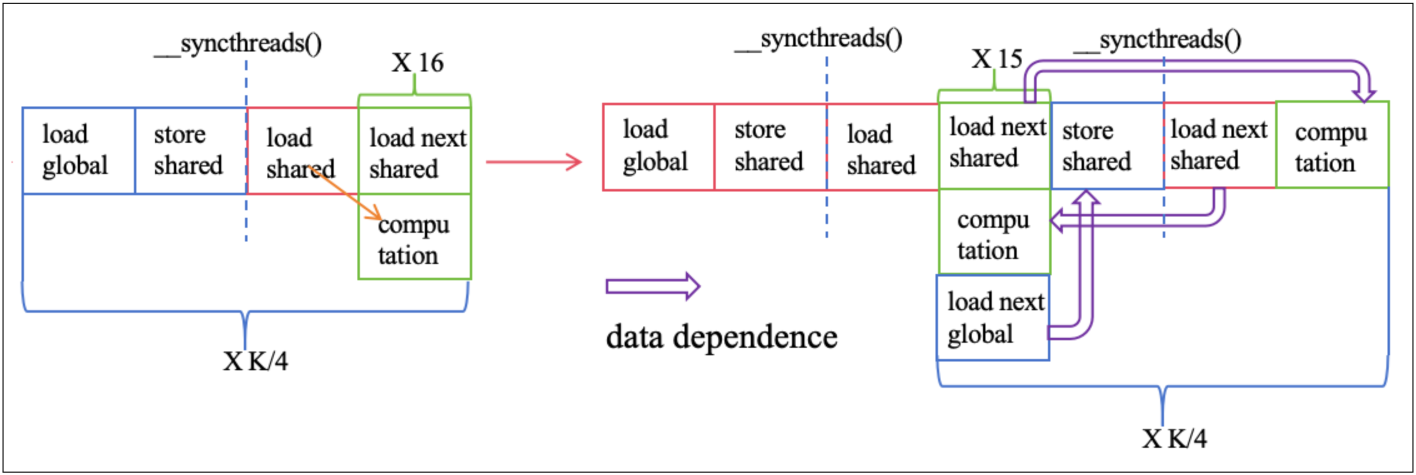
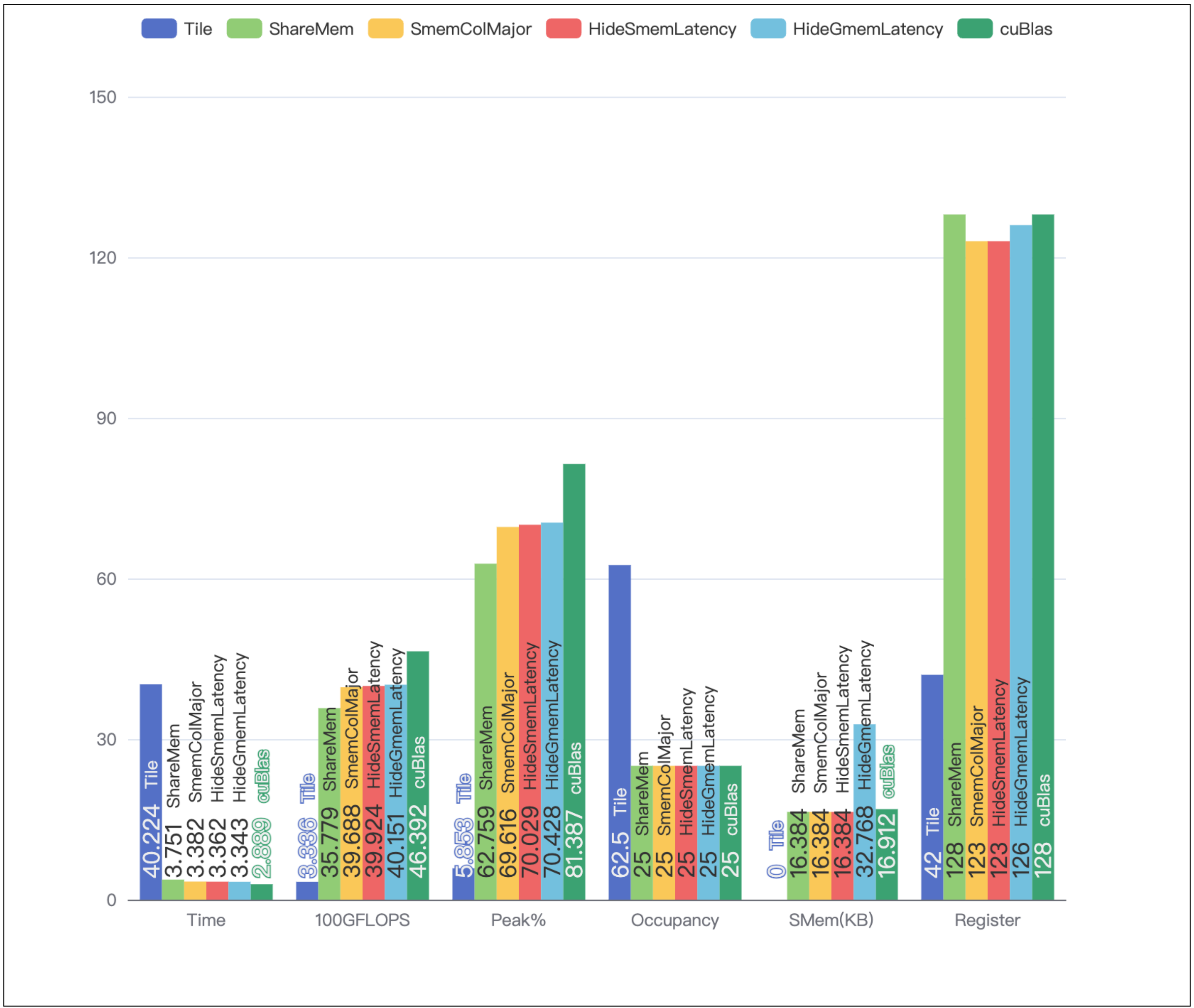


**Fig. 8.** hide global memory IO latency



| | Tile | ShareMem | SmemColMajor | HideSmemLatency | HideGmemLatency | cuBlas |
|---|---|---|---|---|---|---|
| Time | 40.224 | 3.751 | 3.382 | 3.362 | 3.343 | 2.889 |
| 100GFLOPS | 3.336 | 35.779 | 39.688 | 39.924 | 40.151 | 46.392 |
| Peak% | 5.853 | 62.759 | 69.616 | 70.029 | 70.428 | 81.387 |
| Occupancy | 62.5 | 25 | 25 | 25 | 25 | 25 |
| SMem(KB) | 0 | 16.384 | 16.384 | 16.384 | 32.768 | 16.912 |
| Register | 42 | 128 | 123 | 123 | 126 | 128 |

**Fig. 9.** profile